PYTHON

Introduction to Software-defined Networking
Block Course – Winter 2015/16

David Koll

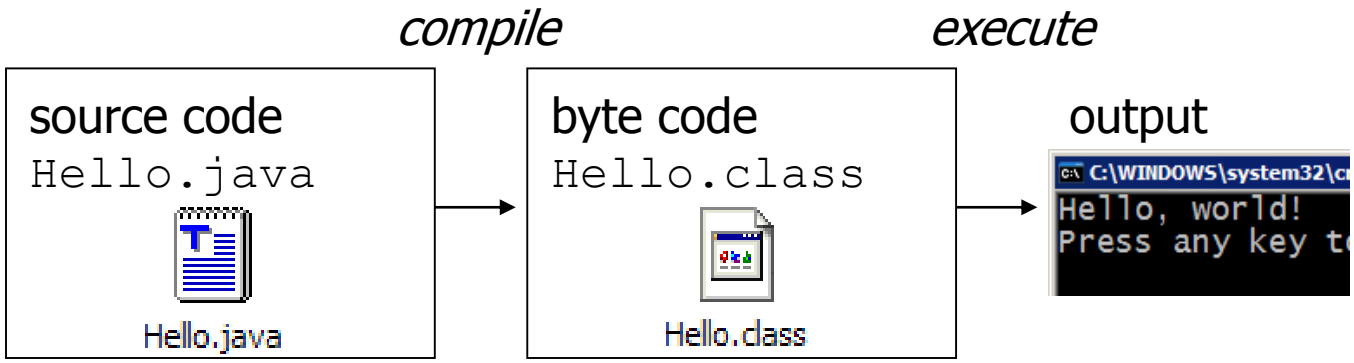
Based on slides of Matt Huenerfauth from the University of Pennsylvania

What is Python?

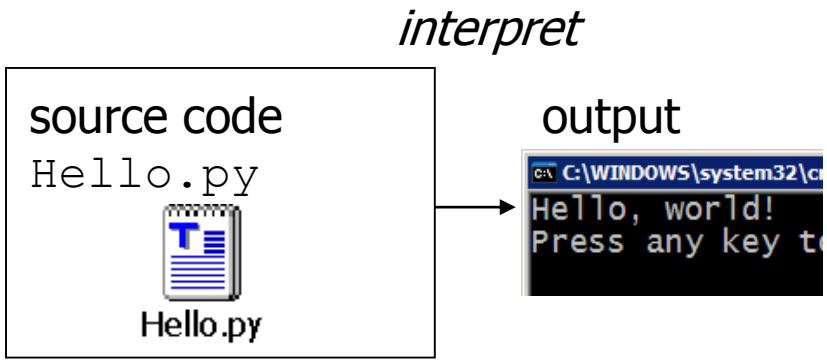
- An *interpreted* programming language
  - …with strong similarities to PERL
  - …with powerful typing and object oriented features.
  - …with useful built-in types (lists, dictionaries).
  - …with clean syntax, powerful extensions.

Compiling and Interpreting

- Many languages require compiling



- Python is directly *interpreted* into machine instructions.



Why Python in this course?

- Simple reason: Mininet
- Mininet is a emulation/virtualization tool to develop, test and deploy (software-defined) networks
- Written in…Python
  - Has an API that you have to understand for this course!

How will we teach Python?

- We expect that you are familiar with object oriented programming
  - We expect that you know at least one of C++/JAVA…
  - …and how to use a development DIE (Eclipse, Emacs, …)
- We expect that you know the concepts of:
  - Data types
  - Variable declaration and assignment
  - Control sequences (if/else, while, for, …)
  - Functions
  - Classes
- We will focus on a recap of the syntax and special issues of Python here!

Technical Issues

Installing & Running Python

Some words ahead

- We will be using Python 2.7
  - For differences to Python 3 (newer version), read online
  - In short:
    - 2.7 is default version on most UNIX OS distributions (e.g. Ubuntu 14)
    - 2.7 has better library support
  - Note that Python 3 will be the future standard (but differences are not that big)

Installing Python

- Python for Windows from www.python.org.
- For UNIX: No installation should be required
- GUI development environments:
  - Eclipse (PyDev)
  - Emacs
  - IDLE.
  - Your favorite IDE or text editor!

Running Interactively on UNIX

On Unix…

```
% python

>>> 3+3
6
```

The '>>>' is the Python prompt.

Running Programs on UNIX

```
% python filename.py
```

You can create python files using emacs.

(There's a special Python editing mode.)

Want to make *.py file executable? Add on top:
#!/pkg/bin/python

Understanding the Basics

Whitespace

- Whitespace is meaningful in Python: ***especially indentation and placement of newlines.***
  - Use a newline to end a line of code. *(Not a semicolon like in C++ or Java.)*
    - (Use \ when must go to next line prematurely.)
  - No braces { } to mark blocks of code in Python…
    - Use consistent *indentation* instead.
    - The first line with a new indentation is considered outside of the block.
  - Often a colon appears at the start of a new block.

Comments

- Start comments with # – the rest of line is ignored.
- Convention: “documentation string”
  - in the first line of any new function or class that you define.

```
def my_function(x, y):
    """This is the docstring. This
    function does blah blah blah."""
    # The code would go here...
```

UNDERSTANDING ASSIGNMENT

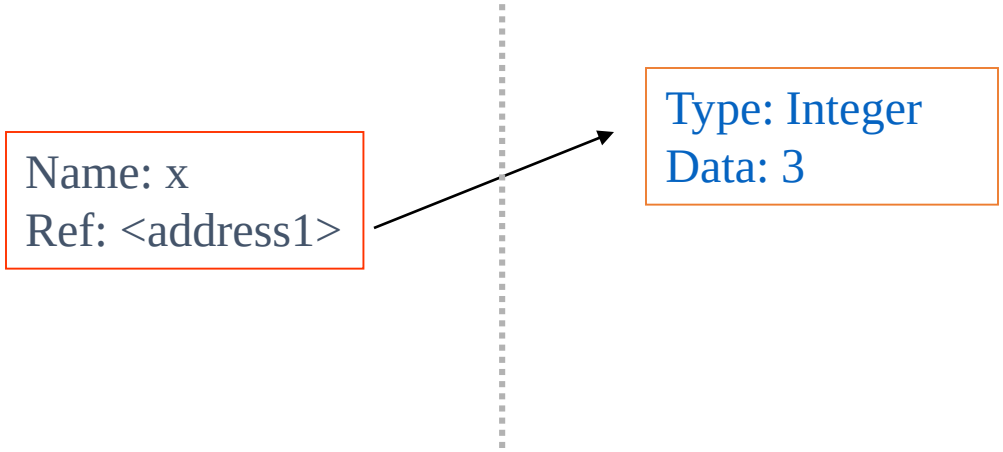
Names and References 3

- In Python, the basic datatypes integer, float, and string are “immutable.”
- This doesn’t mean we can’t change the value of x… For example, we could increment x.

```
>>> x = 3
>>> x = x + 1
>>> print x
4
```

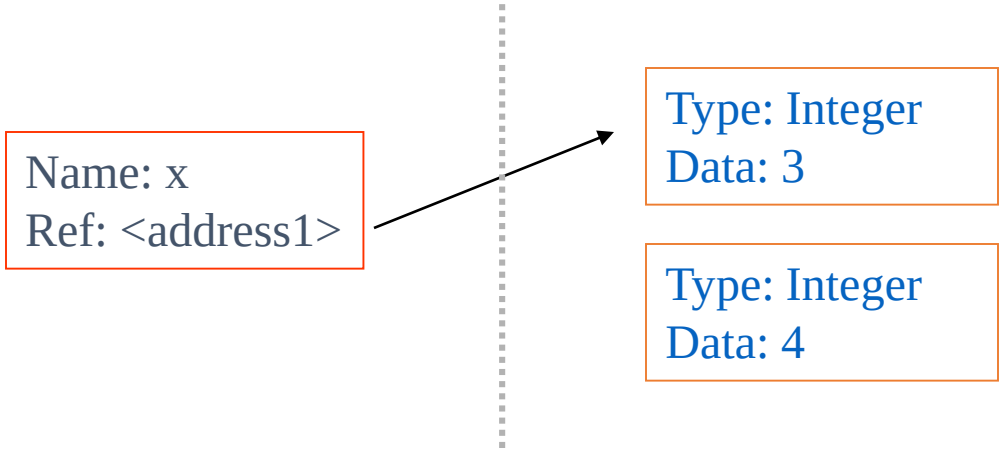
Names and References 4

- If we increment x, then what's really happening is:
  - The reference of name x is looked up.
  - The value at that reference is retrieved.
  - The 3+1 calculation occurs, producing a new data element 4 which is assigned to a fresh memory location with a new reference.
  - The name x is changed to point to this new reference.
  - The old data 3 is garbage collected if no name still refers to it.



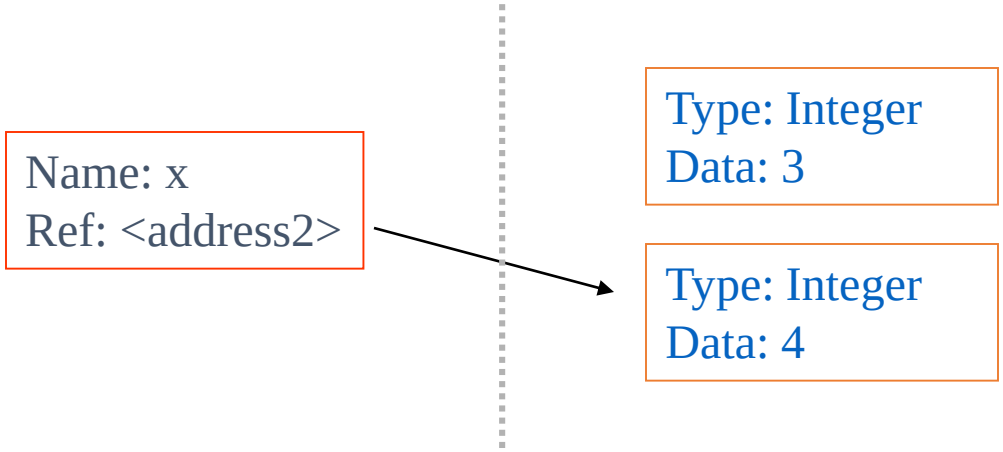
Names and References 4

- If we increment x, then what's really happening is:
  - The reference of name x is looked up.
  - The value at that reference is retrieved.
  - The 3+1 calculation occurs, producing a new data element 4 which is assigned to a fresh memory location with a new reference.
  - The name x is changed to point to this new reference.
  - The old data 3 is garbage collected if no name still refers to it.



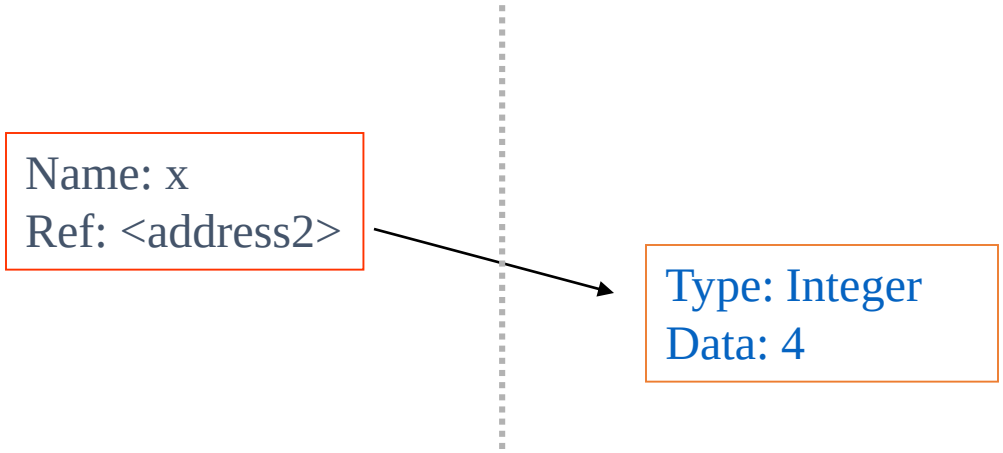
Names and References 4

- If we increment x, then what's really happening is:
  - The reference of name x is looked up.
  - The value at that reference is retrieved.
  - The 3+1 calculation occurs, producing a new data element 4 which is assigned to a fresh memory location with a new reference.
  - The name x is changed to point to this new reference.
  - The old data 3 is garbage collected if no name still refers to it.



Names and References 4

- If we increment x, then what's really happening is:
  - The reference of name x is looked up.
  - The value at that reference is retrieved.
  - The 3+1 calculation occurs, producing a new data element 4 which is assigned to a fresh memory location with a new reference.
  - The name x is changed to point to this new reference.
  - The old data 3 is garbage collected if no name still refers to it.



Assignment 2

- For other data types (lists, dictionaries, user-defined types), assignment works differently.
  - These datatypes are **"mutable."**
  - When we change these data, we do it *in place.*
  - We don't copy them into a new memory address each time.
  - If we type y=x and then modify y, both x and y are changed!
  - We'll talk more about "mutability" later.

immutable

```
>>> x = 3
>>> y = x
>>> y = 4
>>> print x
3
```

mutable

x = some mutable object

y = x

make a change to y

look at x

x will be changed as well

Container Types in Python

Container Types

- Containers are built-in data types in Python.
  - Can hold objects of any type (including their own type).
  - There are three kinds of containers:

Tuples

- A simple immutable ordered sequence of items.

Lists

- Sequence with more powerful manipulations possible.

Dictionaries

- A look-up table of key-value pairs.

Contents can be of varying type!

Lists

- Lists are defined using square brackets (and commas).

```
>>> li = [“abc”, 34, 4.34, 23]
```

- Strings are defined using quotes (“, ‘, or “““).

```
>>> st = “Hello World”
>>> st = ‘Hello World’
>>> st = “““This is a multi-line
string that uses triple quotes.”””
```

Slicing: Return Copy of a Subset 1

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Return a copy of the container with a subset of the original members. Start copying at the first index, and stop copying before the second index.

```
>>> t[1:4]
('abc', 4.56, (2,3))
```

You can also use negative indices when slicing.

```
>>> t[1:-1]
('abc', 4.56, (2,3))
```

Slicing: Return Copy of a Subset 2

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Omit the first index to make a copy starting from the beginning of the container.

```
>>> t[:2]
(23, 'abc')
```

Omit the second index to make a copy starting at the first index and going to the end of the container.

```
>>> t[2:]
(4.56, (2,3), 'def')
```

Copying the Whole Container

You can make a copy of the whole tuple using `[:]`.

```
>>> t[:]
(23, 'abc', 4.56, (2,3), 'def')
```

So, there's a difference between these two lines:

```
>>> list2 = list1   # 2 names refer to 1 ref
                    # Changing one affects both

>>> list2 = list1[:]   # Two copies, two refs
                       # They're independent
```

The ‘in’ Operator

- Boolean test whether a value is inside a container:

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
 False
>>> 4 in t
 True
>>> 4 not in t
 False
```

- Be careful: the ‘in’ keyword is also used in the syntax of other unrelated Python constructions: “for loops” and “list comprehensions.”

Operations on Lists Only 1

- Many more operations we can perform on (mutable) lists than on (immutable) tuples.

- But: lists not as fast as tuples.
  - Trade-off.

Operations on Lists Only 2

```
>>> li = [1, 2, 3, 4, 5]

>>> li.append('a')
>>> li
[1, 2, 3, 4, 5, 'a']

>>> li.insert(2, 'i')
>>>li
[1, 2, 'i', 3, 4, 5, 'a']
```

Operations on Lists Only 3

- ‘+’ vs ‘extend’?
  - + creates a fresh list (with a new memory reference)
  - extend operates on list **`li`** in place.

```
>>> li.extend([9, 8, 7])
>>>li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7]
```

- Extend takes a list as an argument. Append takes a singleton.

```
>>> li.append([9, 8, 7])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7, [9, 8, 7]]
```

Operations on Lists Only 4

```
>>> li = ['a', 'b', 'c', 'b']

>>> li.index('b')      # index of first occurrence
1

>>> li.count('b')      # number of occurrences
2

>>> li.remove('b')     # remove first occurrence
>>> li
  ['a', 'c', 'b']
```

Operations on Lists Only 5

```
>>> li = [5, 2, 6, 8]

>>> li.reverse()     # reverse the list *in place*
>>> li
  [8, 6, 2, 5]

>>> li.sort()        # sort the list *in place*
>>> li
  [2, 5, 6, 8]
```

Tuples vs. Lists

- Lists slower but more powerful than tuples.
  - Lists can be modified, and they have lots of handy operations we can perform on them.
  - Tuples are immutable and have fewer features.

- We can always convert between tuples and lists using the list() and tuple() functions.

```
li = list(tu)
tu = tuple(li)
```

Generating Lists using "List Comprehensions"

List Comprehensions

- A powerful feature of the Python language.
  - Generate a new list by applying a function to every member of an original list.

- The syntax of a “list comprehension” is different from what you may have seen so far

List Comprehensions Syntax 1

```
>>> li = [3, 6, 2, 7]
>>> [elem*2 for elem in li]
[6, 12, 4, 14]
```

[expression **for** name **in** list]

- Where expression is some calculation or operation acting upon the variable name.

- Results are stored in a new list!

List Comprehension Syntax 2

- Also possible on containers:

```
>>> li = [('a', 1), ('b', 2), ('c', 7)]
>>> [ n * 3 for (x, n) in li]
[3, 6, 21]
```

List Comprehension Syntax 3

- The expression of a list comprehension could also contain user-defined functions.

```
>>> def subtract(a, b):
        return a - b

>>> oplist = [(6, 3), (1, 7), (5, 5)]

>>> [subtract(y, x) for (x, y) in oplist]
[-3, 6, 0]
```

Filtered List Comprehension 1

`[` expression `for` name `in` list `if` filter `]`

- Exclude some list members from applying comprehension

- First check each member of the list to see if it satisfies a filter condition.

Filtered List Comprehension 2

```
[ expression for name in list if filter ]
```

```
>>> li = [3, 6, 2, 7, 1, 9]
>>> [elem * 2 for elem in li if elem > 4]
[12, 14, 18]
```

- Only 6, 7, and 9 satisfy the filter condition.

Nested List Comprehensions

- Nested comprehensions possible.

```
>>> li = [3, 2, 4, 1]
>>> [elem*2 for elem in
        [item+1 for item in li] ]
[8, 6, 10, 4]
```

- The inner comprehension produces: [4, 3, 5, 2].
- The outer comprehension produces: [8, 6, 10, 4].

DICTIONARIES

Basic Syntax for Dictionaries 1

- Dictionaries store a mapping between a set of keys and a set of values.
  - Keys can be any immutable type.
  - Values can be any type, ***and you can have different types of values in the same dictionary.***

- You can define, modify, view, lookup, and delete the key-value pairs in the dictionary.

Basic Syntax for Dictionaries 2

```
>>> d = {'user':'bozo', 'pswd':1234}

>>> d['user']
'bozo'

>>> d['pswd']
1234

>>> d['bozo']

Traceback (innermost last):
  File '<interactive input>' line 1, in ?
KeyError: bozo
```

Basic Syntax for Dictionaries 3

```
>>> d = {'user':'bozo', 'pswd':1234}

>>> d['user'] = 'clown'
>>> d
{'user':'clown', 'pswd':1234}
```

Note: Keys are unique.
Assigning to an existing key just replaces its value.

```
>>> d['id'] = 45
>>> d
{'user':'clown', 'id':45, 'pswd':1234}
```

Note: Dictionaries are unordered.
New entry might appear anywhere in the output.

Basic Syntax for Dictionaries 4

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}

>>> del d['user']                    # Remove one.
>>> d
{'p':1234, 'i':34}

>>> d.clear()                        # Remove all.
>>> d
{}
```

Basic Syntax for Dictionaries 5

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}

>>> d.keys()                    # List of keys.
['user', 'p', 'i']

>>> d.values()                  # List of values.
['bozo', 1234, 34]

>>> d.items()          # List of item tuples.
[('user','bozo'), ('p',1234), ('i',34)]
```

Functions in Python

range() function

- range() returns a list of numbers from 0 up to the number we pass to it (non-inclusive).
- range(5) returns [0,1,2,3,4]

```
for x in range(5):
    print x
```

- range([start], stop[, step])

```
range(4:10:2) returns 4,6,8
```

- Can also go backwards (`range(4:-4:-2)`)

Defining Functions

Function definition begins with "def."

Function name and its arguments.

```
def get_final_answer(filename):
    "Documentation String"
    line1
    line2
    return total_counter
```

Colon.

The indentation matters…

The keyword 'return' indicates the value to be sent back to the caller.

No header file or declaration of types of function or arguments.

Calling a Function

- The syntax for a function call is:

```
>>> def myfun(x, y):
        return x * y
>>> myfun(3, 4)
12
```

- Parameters in Python are “Call by Assignment.”
  - Sometimes acts like “call by reference” and sometimes like “call by value” in C++.
    - Mutable datatypes: Call by reference.
    - Immutable datatypes: Call by value.

Functions without returns

- ***All*** functions in Python have a return value
- Functions without a “return” will give the special value **None** as their return value.
  - None is used like NULL, void, or nil in other languages.
  - Also logically equivalent to False.

Function overloading? No.

- There is no function overloading in Python.
  - Unlike C++, a Python function is specified by its name alone
  - the number, order, names, or types of its arguments cannot be used to distinguish between two functions with the same name.

You can't have two functions with the same name, even if they have different arguments.

Treating Functions Like Data

- Functions are treated like first-class objects in the language… They can be passed around like other data and be arguments or return values of other functions.

```
>>> def myfun(x):
        return x*3

>>> def applier(q, x):
        return q(x)

>>> applier(myfun, 7)
21
```

Assignment & Mutability

- When passing parameters to functions:
  - Immutable data types are **"call by value."**
  - Mutable data types are **"call by reference."**

If you pass mutable data to a function, and you change it inside that function, the changes will persist after the function returns.

Immutable data appear unchanged inside of functions to which they are passed.

Some Fancy Function Syntax

Lambda Notation – Anonymous Functions

- Sometimes it is useful to define short functions without having to give them a name: especially when passed as an argument to another function.

```
>>> applier(lambda z: z * 4, 7)
28
```

  - First argument to applier() is an unnamed function that takes one input and returns the input multiplied by four.
  - Note: only single-expression functions can be defined using this lambda notation.

Default Values for Arguments

- Give default values for a function's arguments when defining it
  - these arguments are optional when function is called.

```
>>> def myfun(b, c=3, d="hello"):
        return b + c
>>> myfun(5,3,"hello")
>>> myfun(5,3)
>>> myfun(5)
```

All of the above function calls return 8.

The Order of Arguments

```
>>> def myfun(a, b, c):
        return a-b
>>> myfun(2, 1, 43)
  1
>>> myfun(c=43, b=1, a=2)
  1
>>> myfun(2, c=43, b=1)
  1
```

Defining Classes

Defining a Class

- Python doesn't use separate class interface definitions as in some languages.
  - You just define the class and then use it.

- You can define a method in a class by including function definitions within the scope of the class block.
  - Take care of proper indentation!

Definition of student

```
#class definition
class student:
    """A class representing a
    student."""
    #init function?
    #self argument?
    def __init__(self,n,a):
        self.full_name = n
        self.age = a

    def get_age(self):
        return self.age
```

Constructor: __init__

- **`__init__`** acts like a constructor for a class.
  - Invoked upon instantiating a class
  - **`b = student("Bob", 21)`**

    __init__ is passed “Bob” and 21.

Constructor: __init__

- **`__init__`** can take any number of arguments.

- However, the first argument **`self`** in the definition of __init__ is special…

Self

- The first argument of every class method is a reference to the current instance of the class.
  - By convention, this argument is named **`self`**.

- In **`__init__`**, self refers to the object currently being created
- in other class methods, it refers to the instance whose method was called.

  - Similar to the keyword ‘this’ in Java or C++.
  - But Python uses ‘self’ more often than Java uses ‘this.’

Self

- Although you must specify **`self`** explicitly when defining the method, you don't include it when calling the method.
- Python passes it for you automatically.

Defining a method:

(this code inside a class definition.)

```
def set_age(self, num):
  self.age = num
```

Calling a method:

```
>>> x.set_age(23)
```

Creating and Deleting Instances

Instantiating Objects

- There is no “new” keyword as in Java.
- You merely use the class name with () notation and assign the result to a variable.

```
b = student(“Bob Smith”, 21)
```

- The arguments you pass to the class name are actually given to its `.__init__()` method.

- User-defined classes are mutable!

No Need to “free”

- When you are done with an object, you don’t have to delete or free it explicitly.
  - Python has automatic garbage collection.
  - Generally works well, few memory leaks.
  - There’s also no “destructor” method for classes.

Access to Attributes and Methods

Definition of student

```
class student:
 """A class representing a
 student."""
 def __init__(self,n,a):
      self.full_name = n
      self.age = a
 def get_age(self):
      return self.age
```

Traditional Syntax for Access

```
>>> f = student ("Bob Smith", 23)

>>> f.full_name      # Access an attribute.
"Bob Smith"

>>> f.get_age()      # Access a method.
23
```

Attributes

Two Kinds of Attributes

- The non-method data stored by objects are called attributes. There's two kinds:
  - **Data attribute:**
    - Variable owned by a particular instance of a class.
  - **Class attributes:**
    - Owned by the class as a whole.
    - Called "static" variables in some languages.
    - Good for class-wide constants or for building counter of how many instances of the class have been made.

Data Attributes

- inside of the **`__init__()`** method.
  - Inside the class, refer to data attributes using **`self`** – for example, **`self.full_name`**

```
class teacher:
    "A class representing teachers."
    def __init__(self,n):
        self.full_name = n
    def print_name(self):
        print self.full_name
```

Class Attributes

- All instances of a class share one copy of a class attribute
  - if any of the instances changes it, value is changed for all instances.
- Define class attributes outside of any method.
- Access them using **`self.__class__.name`** notation.

```
class sample:
x = 23
def increment(self):
    self.__class__.x += 1
```

```
>>> a = sample()
>>> a.increment()
>>> a.__class__.x
24
```

Data vs. Class Attributes

```
class counter:
 overall_total = 0
       # class attribute
 def __init__(self):
      self.my_total = 0
        # data attribute
 def increment(self):
     counter.overall_total = \
     counter.overall_total + 1
     self.my_total = \
     self.my_total + 1
```

```
>>> a = counter()
>>> b = counter()
>>> a.increment()
>>> b.increment()
>>> b.increment()
>>> a.my_total
1
>>> a.__class__.overall_total
3
>>> b.my_total
2
>>> b.__class__.overall_total
3
```

INHERITANCE

Subclasses

- Inheritance works pretty much like in other languages
  - New class: “subclass.” Original: “parent” or “ancestor.”
- Syntax of defining a subclass:

```
class subclass(parent):
```

e.g.:

```
class student(person):
```

- Python has no ‘extends’ keyword like Java.
- Multiple inheritance is supported.

Definition of student

```
class person:
 "A class representing a person."

 def __init__(self,n,a):
     self.full_name = n
     self.age = a

 def get_age(self):
     return self.age
```

Definition of ai_student

```
class ai_student (person):
 "A class extending person."

 def __init__(self,n,a,number):
      student.__init__(self,n,a)
      self.mat_num = number

 def get_age():
      print "Age: " + str(self.age)
```

Redefining Methods

- Overwrite parent class methods if desired.

- Can still explicitly call the parent's version of the method.

```
parentClass.methodName(self, a, b, c)
```

The only time you ever explicitly pass 'self' as an argument is when calling a method of an ancestor.

Redefining the __init__

- Same as for redefining any other method…
  - You'll often see something like this in the **`__init__`** method of subclasses:

```
parentClass.__init__(self, x, y)
```

Private Data and Methods

- Any attribute or method with two **leading underscores** is private.
  - Note:
    Names with two underscores **at the beginning and the end** are for built-in methods or attributes for the class.
  - Note:
    There is no ‘protected’ status in Python; so, subclasses would be unable to access these private data either.

Importing and Modules

Importing and Modules

- Use classes & functions defined in another file.
- Like Java import, C++ include.
- Three formats of the command:

```
import somefile
from somefile import *
from somefile import className
```

What's the difference?

What gets imported from the file and what name you use to refer to it after its been imported.

File Handling

inflobj = open(‘data’, ‘r’)	Open the file ‘data’ for input
S = inflobj.read()	Read whole file into one String
S = inflobj.read(N)	Reads N bytes (N >= 1)
L = inflobj.readlines()	Returns a list of line strings

Files: Output

outflobj = open(‘data’, ‘w’)	Open the file ‘data’ for writing
outflobj.write(S)	Writes the string S to file
outflobj.writelines(L)	Writes each of the strings in list L to file
outflobj.close()	Closes the file

Example: Read a file and print line by line

```
fileptr = open('filename')
somestring = fileptr.read()
for line in fileptr:
    print line
fileptr.close()
```

Remember to close all opened files!

Finally: Error Handling

Exception Handling

- Errors are a kind of object in Python.
  - More specific kinds of errors are subclasses of the general Error class.

- You use the following commands to interact with them:
  - Try
  - Except
  - Finally
  - Catch